

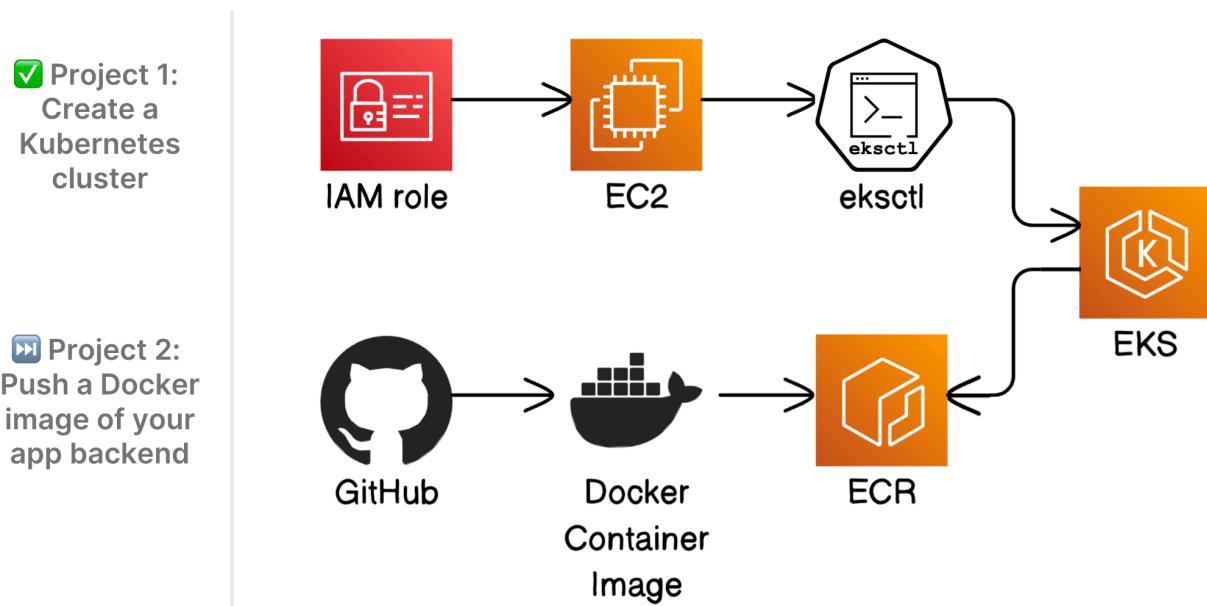
Cloud-Native Backend Deployment with Docker, ECR, and Amazon EKS

Prepared an app's backend for deployment with Kubernetes!

🎯 Project Objective

The objective of this project was to containerize a backend application and deploy it on a Kubernetes cluster using AWS services. The backend code was cloned from a GitHub repository, and a Docker image was built and pushed to Amazon Elastic Container Registry (ECR). The Docker image was then deployed to an Amazon Elastic Kubernetes Service (EKS) cluster. This project enhanced my skills in containerization, AWS ECR, and Kubernetes orchestration, while also reinforcing best practices for cloud-native deployments.

Architecture Overview





Tech Stack

- **Git & GitHub** – Cloned the backend code and tracked changes
- **Docker** – Containerized the backend application
- **Amazon ECR** – Stored the Docker image for deployment
- **Amazon EKS** – Deployed the containerized app using Kubernetes
- **Amazon EC2** – Used as a management server to run commands
- **IAM** – Managed access and permissions for AWS services
- **eksctl** – Created and managed the EKS cluster
- **kubectl** – Deployed and managed Kubernetes resources on EKS

Step 1: Set up EC2 and EKS

Environment Setup

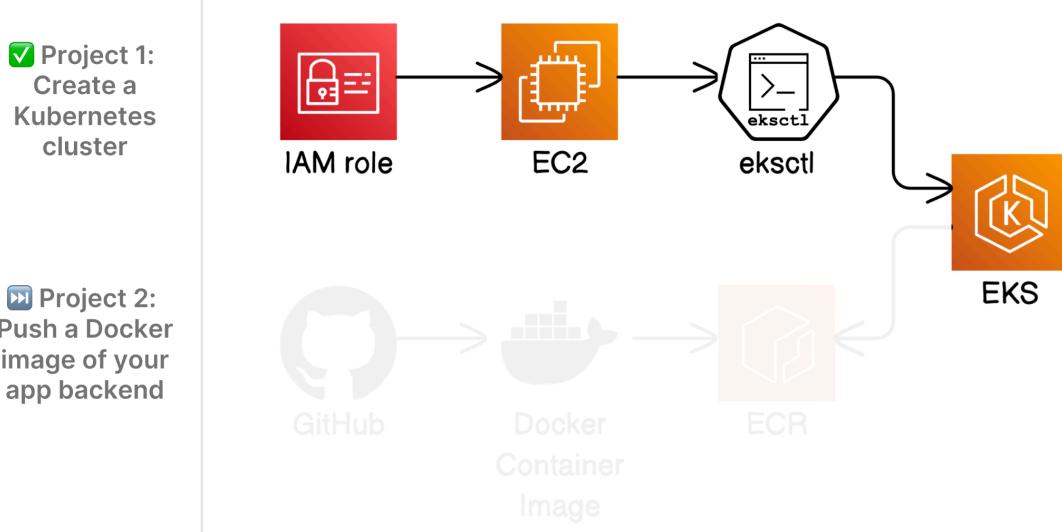
To begin the project, I set up the environment by launching an **Amazon EC2** instance. This instance served as my control machine, where I installed the necessary tools and executed all commands to provision and manage the Kubernetes infrastructure on AWS.

What is Amazon EKS?

Amazon Elastic Kubernetes Service (EKS) is a managed Kubernetes service by AWS that simplifies deploying, managing, and scaling containerized applications. It handles complex tasks like networking, security, and cluster management, allowing users to run Kubernetes without manually setting up the infrastructure.

In this step, I:

- Launched and connected to an Amazon EC2 instance
- Set up **eksctl** and configured the instance with my AWS credentials
- Created an Amazon EKS cluster to manage my Kubernetes workloads



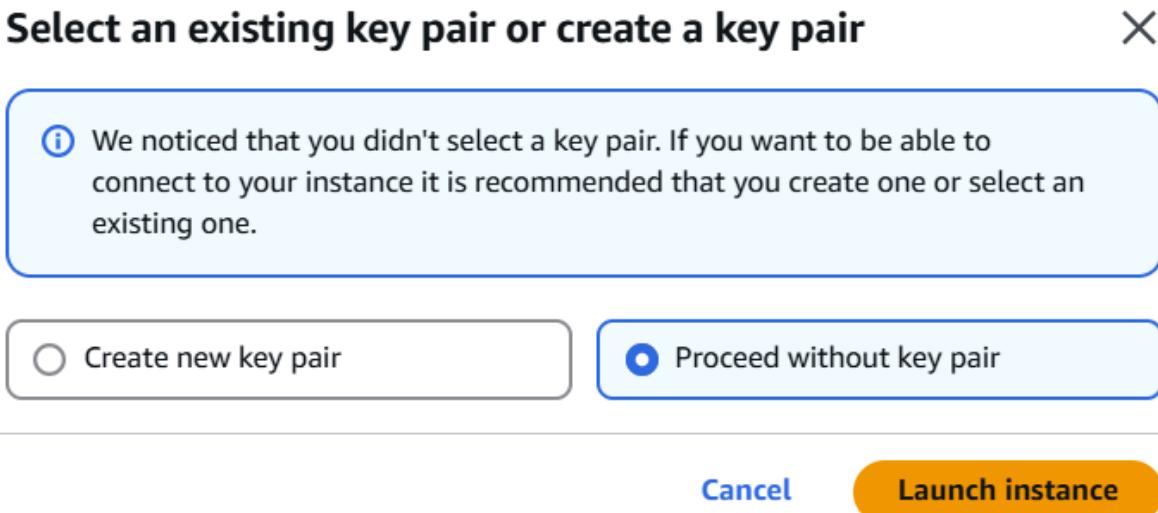
To get started, I navigated to the EC2 console in my AWS account.

- I made sure to select the AWS Region closest to my location for better performance.
- I clicked “**Launch instances**” to create a new virtual machine.
- I named my instance **dschundru-eks-instance** for easy identification.
- For the **Amazon Machine Image (AMI)**, I selected Amazon Linux 2023 AMI, which is optimized for compatibility with AWS tools and services.

The screenshot shows the AWS EC2 'Launch an instance' wizard. The 'Name and tags' step has the instance named 'dschundru-eks-instance'. The 'Application and OS Images (Amazon Machine Image)' step shows the 'Amazon Linux 2023 AMI' selected from a catalog. The 'Amazon Machine Image (AMI)' details page shows the selected AMI information: 'Amazon Linux 2023 AMI' (ami-060a84cbc5c14844), 'Free tier eligible', and 'Virtualization: hvm ENA enabled: true Root device type: ebs'.

I chose **t2.micro** as the instance type since it's free-tier eligible. For the **network settings**, I kept the **default security group** for now.

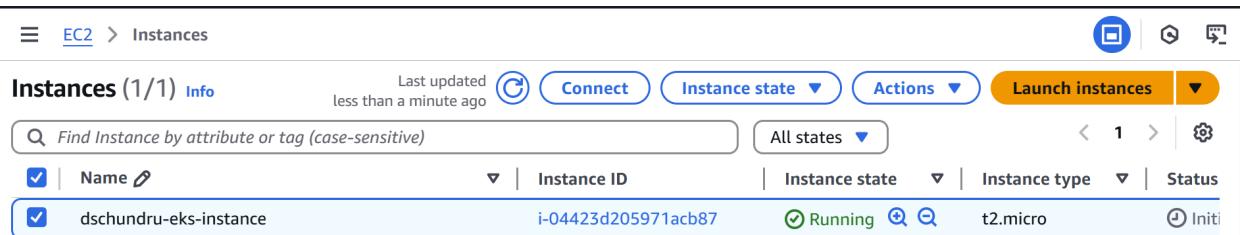
- Select Launch instance and Proceed without a key pair (not recommended).



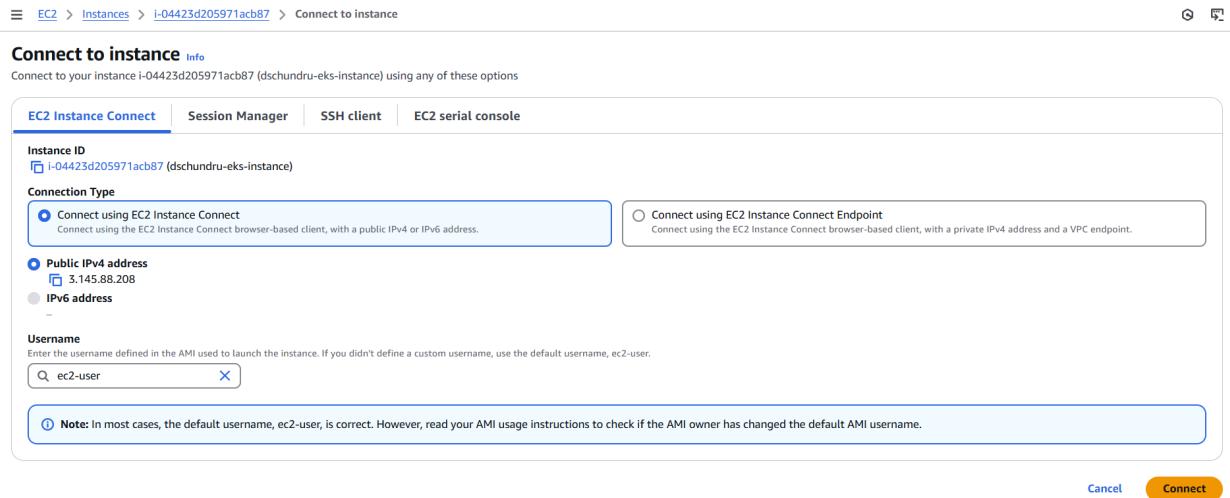
I clicked **Launch instance** to start the setup.



Once the instance appeared on the **Instances** page, I selected it and clicked **Connect**.



In the **EC2 Instance Connect** section, I used the browser-based option and clicked **Connect** again to access the instance directly.



Installed eksctl

eksctl is a command-line tool for managing Amazon EKS clusters.

I'm using eksctl because it's one of the easiest ways to create and manage EKS clusters. It saves me time by reducing the number of commands I need to run. eksctl automatically handles things like setting up networking resources, which makes the whole setup process much smoother.

I ran the following command in my EC2 instance's terminal to download, extract, and install eksctl:

```
[ec2-user@ip-172-31-9-43 ~]$ curl --silent --location "https://github.com/weaveworks/eksctl/releases/latest/download/eksctl_$(uname -s)_amd64.tar.gz" | tar xz -C /tmp
sudo mv -v /tmp/eksctl /usr/local/bin
copied '/tmp/eksctl' -> '/usr/local/bin/eksctl'
removed '/tmp/eksctl'
[ec2-user@ip-172-31-9-43 ~]$ ]
```

- Verified that I've installed eksctl

```
[ec2-user@ip-172-31-9-43 ~]$ eksctl version
0.207.0
```

Created an IAM role for my EC2 Instance

I went to the IAM console, selected **Roles**, clicked **Create role**, and chose **AWS service** as the trusted entity type for Amazon EC2.

Why do I need to set up an IAM role?

I set up an IAM role because my EC2 instance doesn't have any permissions by default. Without it, eksctl can't create EKS clusters. The role lets my instance access AWS services it needs.

Use case
Allow an AWS service like EC2, Lambda, or others to perform actions in this account.

Service or use case
EC2

Choose a use case for the specified service.

Use case

- EC2**
Allows EC2 instances to call AWS services on your behalf.
- EC2 Role for AWS Systems Manager**
Allows EC2 instances to call AWS services like CloudWatch and Systems Manager on your behalf.
- EC2 Spot Fleet Role**
Allows EC2 Spot Fleet to request and terminate Spot Instances on your behalf.
- EC2 - Spot Fleet Auto Scaling**
Allows Auto Scaling to access and update EC2 spot fleets on your behalf.
- EC2 - Spot Fleet Tagging**
Allows EC2 to launch spot instances and attach tags to the launched instances on your behalf.
- EC2 - Spot Instances**
Allows EC2 Spot Instances to launch and manage spot instances on your behalf.
- EC2 - Spot Fleet**
Allows EC2 Spot Fleet to launch and manage spot fleet instances on your behalf.
- EC2 - Scheduled Instances**
Allows EC2 Scheduled Instances to manage instances on your behalf.

Cancel **Next**

- Selected **Next**. Under **Permissions policies**, I granted my EC2 instance **AdministratorAccess**.

Permissions policies (1/1049) Info



Choose one or more policies to attach to your new role.

Filter by Type

 admin


All types



42 matches



1

2

3


 Policy name

 Type

 AdministratorAccess

AWS managed - job function

I checked AdministratorAccess, clicked Next, named the role **dschundru-eks-instance-role**, and added a short description.

Role details

Role name

Enter a meaningful name to identify this role.

dschundru-eks-instance-role

Maximum 64 characters. Use alphanumeric and '+=@-_` characters.

Description

Add a short explanation for this role.

Grants an EC2 instance AdministratorAccess to my AWS account. Created during Kubernetes project.

Maximum 1000 characters. Use letters (A-Z and a-z), numbers (0-9), tabs, new lines, or any of the following characters: _+=,. @-/\[{\}]!#\$%^*`~;:,`

✓ Role dschundru-eks-instance-role created.

[View role](#)



Attached IAM role to EC2 instance

I went back to the EC2 console, selected Instances, checked my dschundru-eks-instance, clicked Actions > Security > Modify IAM role to attach the IAM role.

The screenshot shows the AWS EC2 Instances page. In the top navigation bar, 'EC2' is selected. Below it, the 'Instances (1/1)' section shows one instance named 'dschundru-eks-instance'. The 'Actions' dropdown menu is open, and the 'Modify IAM role' option is highlighted with a light gray background. Other options in the dropdown include 'Connect', 'View details', 'Manage instance state', 'Instance settings', 'Networking', 'Security', 'Image and templates', and 'Monitor and troubleshoot'.

Under **IAM role**, I selected my **dschundru-eks-instance-role** and clicked **Update IAM role** to attach it to the instance.

The screenshot shows the 'Modify IAM role' page in the AWS IAM console. At the top, it says 'EC2 > Instances > i-04423d205971acb87 > Modify IAM role'. Below that, it says 'Attach an IAM role to your instance.' A dropdown labeled 'Instance ID' contains 'i-04423d205971acb87 (dschundru-eks-instance)'. Under 'IAM role', it says 'Select an IAM role to attach to your instance or create a new role if you haven't created any. The role you select replaces any roles that are currently attached to your instance.' A dropdown menu is open, showing 'dschundru-eks-instance-role'. To the right of the dropdown is a 'Create new IAM role' button. At the bottom are 'Cancel' and 'Update IAM role' buttons.

Created EKS Cluster

I headed back to EC2 Instance Connect and ran the command to create my EKS cluster, replacing **YOUR-REGION** with my region code:

```
eksctl create cluster \
--name dschundru-eks-cluster \
--nodegroup-name dschundru-nodegroup \
--node-type t2.micro \
--nodes 3 \
--nodes-min 1 \
--nodes-max 3 \
--version 1.31 \
--region us-east-2
```

```
[ec2-user@ip-172-31-9-43 ~]$ eksctl create cluster \
--name dschundru-eks-cluster \
--nodegroup-name dschundru-nodegroup \
--node-type t2.micro \
--nodes 3 \
--nodes-min 1 \
--nodes-max 3 \
--version 1.31 \
--region us-east-2
2025-04-25 01:41:23 [i] eksctl version 0.207.0
2025-04-25 01:41:23 [i] using region us-east-2
2025-04-25 01:41:23 [i] setting availability zones to [us-east-2a us-east-2c us-east-2b]
2025-04-25 01:41:23 [i] subnets for us-east-2a - public:192.168.0.0/19 private:192.168.96.0/19
2025-04-25 01:41:23 [i] subnets for us-east-2c - public:192.168.32.0/19 private:192.168.128.0/19
2025-04-25 01:41:23 [i] subnets for us-east-2b - public:192.168.64.0/19 private:192.168.160.0/19
2025-04-25 01:41:23 [i] nodegroup "dschundru-nodegroup" will use "" [AmazonLinux2/1.31]
2025-04-25 01:41:23 [i] using Kubernetes version 1.31
2025-04-25 01:41:23 [i] creating EKS cluster "dschundru-eks-cluster" in "us-east-2" region with managed nodes
2025-04-25 01:41:23 [i] will create 2 separate CloudFormation stacks for cluster itself and the initial managed nodegroup
2025-04-25 01:41:23 [i] if you encounter any issues, check CloudFormation console or try 'eksctl utils describe-stacks --region=us-east-2 --cluster=dschundru-eks-cluster'
2025-04-25 01:41:23 [i] Kubernetes API endpoint access will use default of {publicAccess=true, privateAccess=false} for cluster "dschundru-eks-cluster" in "us-east-2"
2025-04-25 01:41:23 [i] CloudWatch logging will not be enabled for cluster "dschundru-eks-cluster" in "us-east-2"
2025-04-25 01:41:23 [i] you can enable it with 'eksctl utils update-cluster-logging --enable-types=(SPECIFY-YOUR-LOG-TYPES-HERE (e.g. all)) --region=us-east-2 --cluster=dschundru-eks-cluster'
2025-04-25 01:41:23 [i] default addons vpc-cni, kube-proxy, coredns, metrics-server were not specified, will install them as EKS addons
2025-04-25 01:41:23 [i] 2 sequential tasks: { create cluster control plane "dschundru-eks-cluster",
    2 sequential sub-tasks: {
        2 sequential sub-tasks: {
```

```

        wait for control plane to become ready,
    },
    create managed nodegroup "dschundru-nodegroup",
}
}

2025-04-25 01:41:23 [i]   building cluster stack "eksctl-dschundru-eks-cluster-cluster"
2025-04-25 01:41:23 [i]   deploying stack "eksctl-dschundru-eks-cluster-cluster"
2025-04-25 01:41:53 [i]   waiting for CloudFormation stack "eksctl-dschundru-eks-cluster-cluster"
2025-04-25 01:42:23 [i]   waiting for CloudFormation stack "eksctl-dschundru-eks-cluster-cluster"
2025-04-25 01:43:23 [i]   waiting for CloudFormation stack "eksctl-dschundru-eks-cluster-cluster"
2025-04-25 01:44:23 [i]   waiting for CloudFormation stack "eksctl-dschundru-eks-cluster-cluster"
2025-04-25 01:45:23 [i]   waiting for CloudFormation stack "eksctl-dschundru-eks-cluster-cluster"
2025-04-25 01:46:23 [i]   waiting for CloudFormation stack "eksctl-dschundru-eks-cluster-cluster"
2025-04-25 01:47:23 [i]   waiting for CloudFormation stack "eksctl-dschundru-eks-cluster-cluster"
2025-04-25 01:48:24 [i]   waiting for CloudFormation stack "eksctl-dschundru-eks-cluster-cluster"
2025-04-25 01:49:24 [i]   waiting for CloudFormation stack "eksctl-dschundru-eks-cluster-cluster"
2025-04-25 01:49:24 [i]   recommended policies were found for "vpc-cni" addon, but since OIDC is disabled on the cluster, eksctl cannot configure the recommended permissions; the recommended way to provide IAM permissions for "vpc-cni" addon is via pod identity associations; after addon creation is completed add all recommended policies to the config file, under 'addon.PodIdentityAssociations', and run 'eksctl update addon'
2025-04-25 01:49:25 [i]   creating addon: vpc-cni
2025-04-25 01:49:25 [i]   successfully created addon: vpc-cni
2025-04-25 01:49:25 [i]   creating addon: kube-proxy
2025-04-25 01:49:25 [i]   successfully created addon: kube-proxy
2025-04-25 01:49:25 [i]   creating addon: coredns
2025-04-25 01:49:26 [i]   successfully created addon: coredns
2025-04-25 01:49:26 [i]   creating addon: metrics-server
2025-04-25 01:49:26 [i]   successfully created addon: metrics-server
2025-04-25 01:51:26 [i]   building managed nodegroup stack "eksctl-dschundru-eks-cluster-nodegroup-dschundru-nodegroup"
2025-04-25 01:51:27 [i]   deploying stack "eksctl-dschundru-eks-cluster-nodegroup-dschundru-nodegroup"
2025-04-25 01:51:27 [i]   waiting for CloudFormation stack "eksctl-dschundru-eks-cluster-nodegroup-dschundru-nodegroup"
2025-04-25 01:51:57 [i]   waiting for CloudFormation stack "eksctl-dschundru-eks-cluster-nodegroup-dschundru-nodegroup"

```

Step 2: Pull the Code for my Backend

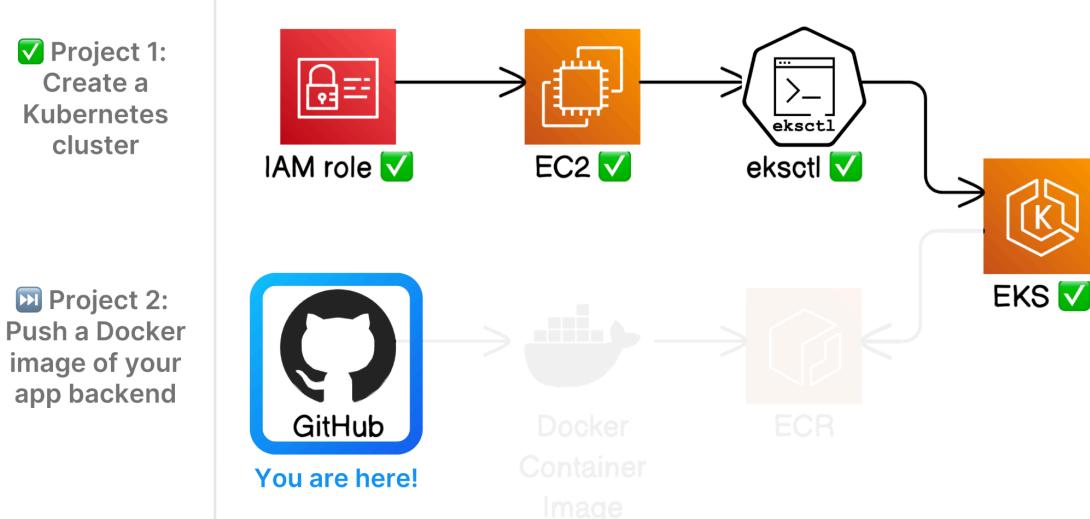
Now that my EKS cluster is spinning up, I pulled the backend code from my team's GitHub repository.

What is 'backend'?

The backend is the brain of the app; it handles requests, stores and fetches data, and makes sure everything works behind the scenes. While the frontend is what users see, the backend powers the logic and functionality.

In this step, I:

- Installed Git and cloned the application code from GitHub to my EC2 instance.



While my cluster was being created, I headed back to the EC2 console and started a new EC2 Instance Connect session with my dschundru-eks-instance-role instance.

```
'          #
~\_\_###_ Amazon Linux 2023
~~ \_\_\#\#\#
~~ \#\#| https://aws.amazon.com/linux/amazon-linux-2023
~~ \#/ V~' '-->
~~~ /
~~ ._. /_
/_m/ . /_
Last login: Fri Apr 25 00:56:54 2025 from 3.16.146.4
[ec2-user@ip-172-31-9-43 ~]$
```

I could connect to the same instance twice at the same time!

- It was true that I was able to open multiple SSH sessions to the same EC2 instance.

This was super helpful when I wanted to multitask or monitor logs while running commands in another session.

Let's clone my backend repository.

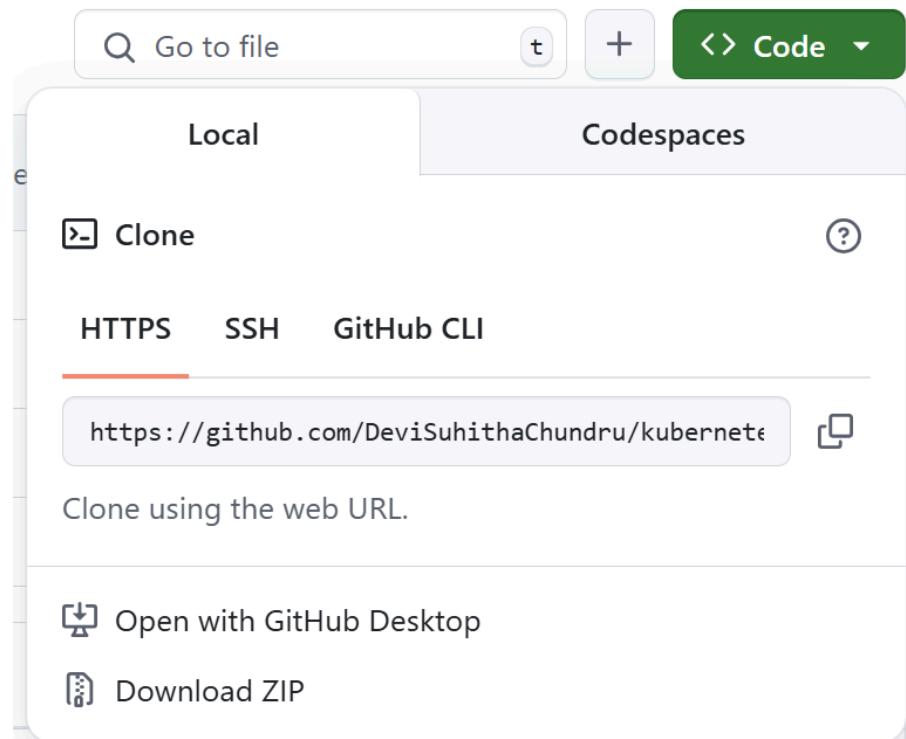
What does cloning mean?

When I say "clone a repository," I mean creating a full copy of a repository's code stored remotely (in this case, in GitHub) to my machine (in this case, our EC2 instance).

This way, I get access to all the files and code without having to manually copy and paste.

The screenshot shows a GitHub repository page for 'kubernetes_part2_dschundru-flask-backend'. The repository is public and has one branch ('main') and one commit ('46130b4 · now'). The README file is present and describes the project as 'The Flask backend project on EKS'. The right sidebar includes sections for 'About', 'Releases', and 'Packages', which are currently empty.

- Selected **Code** to reveal the instructions for cloning their repository and copied the **HTTPS** URL.



I headed back to my **EC2 Instance Connect** tab.

In the terminal, I ran the command below to clone the backend repository, replacing **[GITHUB-URL]** with my own GitHub repo URL:

```
git clone [GITHUB-URL]
```

```
[ec2-user@ip-172-31-9-43 ~]$ git clone https://github.com/DeviSuhithaChundru/Kubernetes-Part-2_dschundru-flask-backend.git
-bash: git: command not found
```

What is this error about?

The terminal response "Git command not found" means I don't have **Git** installed in my EC2 instance. I need to install it first to clone a GitHub repository.

Installed Git:

```
[ec2-user@ip-172-31-9-43 ~]$ sudo dnf update
sudo dnf install git -y
```

```
Installed:
  git-2.47.1-1.amzn2023.0.2.x86_64          git-core-2.47.1-1.amzn2023.0.2.x86_64 git-core
  -doc-2.47.1-1.amzn2023.0.2.noarch perl-Error-1:0.17029-5.amzn2023.0.2.noarch
  perl-File-Find-1.37-477.amzn2023.0.6.noarch perl-Git-2.47.1-1.amzn2023.0.2.noarch perl-Ter
mReadKey-2.38-9.amzn2023.0.2.x86_64 perl-lib-0.65-477.amzn2023.0.6.x86_64
Complete!
```

- I verified that Git was installed by checking its version using the command:

```
[ec2-user@ip-172-31-9-43 ~]$ git --version
git version 2.47.1
```

I configured Git by running the following commands and replacing the placeholders with my actual name and email:

```
git config --global user.name "DeviSuhithaChundru"
git config --global user.email "cdevisuhitha@gmail.com"
```

```
[ec2-user@ip-172-31-9-43 ~]$ git config --global user.name "DeviSuhithaChundru"
git config --global user.email "cdevisuhitha@gmail.com"
```

Now I should be able to clone the repository:

```
[ec2-user@ip-172-31-9-43 ~]$ git clone https://github.com/DeviSuhithaChundru/kubernetes_part2_dschundru-flask-backend.git
Cloning into 'kubernetes_part2_dschundru-flask-backend'...
remote: Enumerating objects: 6, done.
remote: Counting objects: 100% (6/6), done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 6 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
Receiving objects: 100% (6/6), done.
[ec2-user@ip-172-31-9-43 ~]$ ||
```

What was I Cloning?

I am cloning the kubernetes_part2_dschundru-flask-backend repository from GitHub. It has all the backend code I need, like app.py, Dockerfile, and requirements.txt.

By cloning it, I'm copying the entire project to my EC2 instance. After cloning, it will show up as a new folder with all the backend files inside, ready for me to build, run, and deploy.

I ran the **ls** command, which lists all files and folders in my current directory:

ls

This helped me confirm that a new folder named Kubernetes-Part-2_dschundru-flask-backend was successfully created in my EC2 instance after cloning the repository.

```
[ec2-user@ip-172-31-9-43 ~]$ ls
Kubernetes-Part-2_dschundru-flask-backend
```

Why does the repository name say 'flask'?

The repository name includes "**flask**" because the backend code is built using Flask, a lightweight web framework for Python. Developers use Flask to quickly create backend services or APIs.

It's called a **framework** because it provides built-in tools and templates that make it easier to handle things like HTTP requests, connect to databases, and manage routing all without having to build everything from scratch.

Step 3: Built a Container Image for My Backend

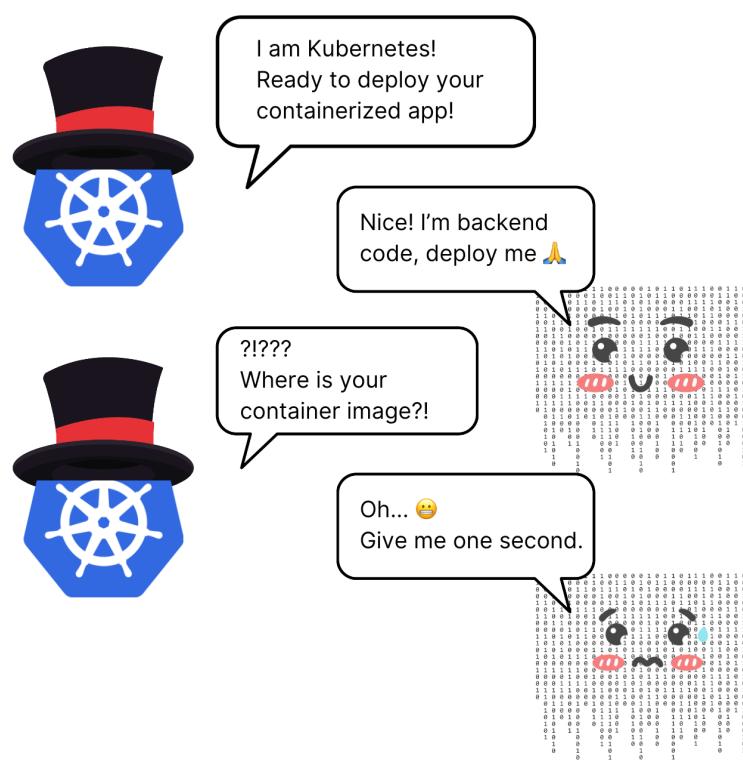
I had my backend code sitting in my EC2 instance.

On to deployment?!

When I deployed a containerized app to Kubernetes, Kubernetes needed to pull the app from a container image that it could access.

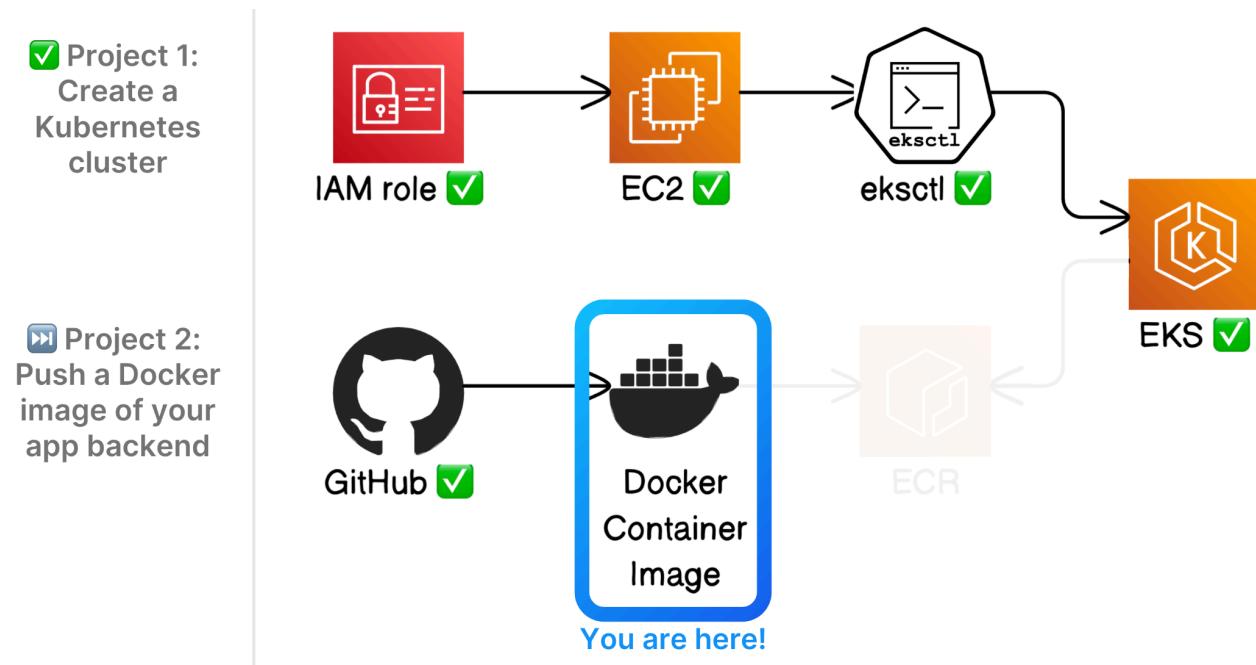
What is a container image?

A **container image** is like a blueprint that contains all the instructions, code, libraries, and dependencies needed to run the application. Note that on the other hand, a **container** is the running instance created from that image, bringing the application to life and running it in an environment.



In this step, I:

- built a container image of the backend and resolved four **installation** and **configuration errors**.



To build a **container image**, I ran the following command in my EC2 instance's terminal:

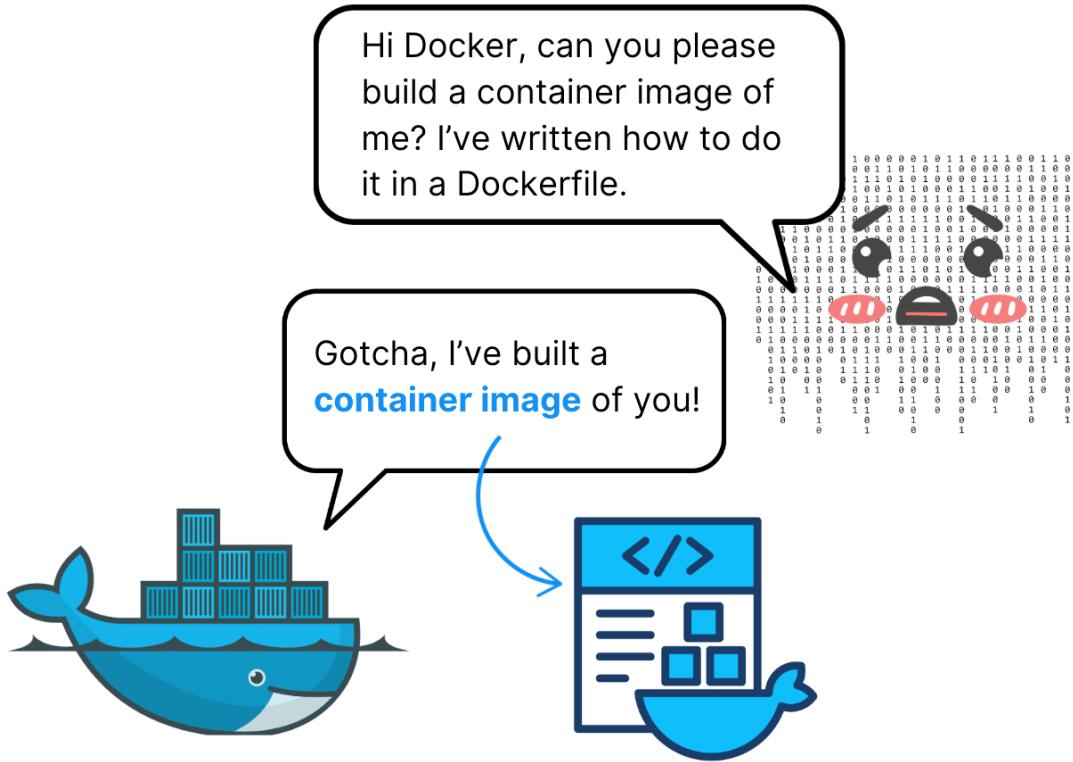
```
docker build -t kubernetes-part2_dschundru-flask-backend .
```

What did building an image mean for me?

When my team members prepared the app's backend, they included a file called a **Dockerfile** in the GitHub repository. This Dockerfile had step-by-step instructions for how to build a container image that packages the backend app along with all its dependencies.

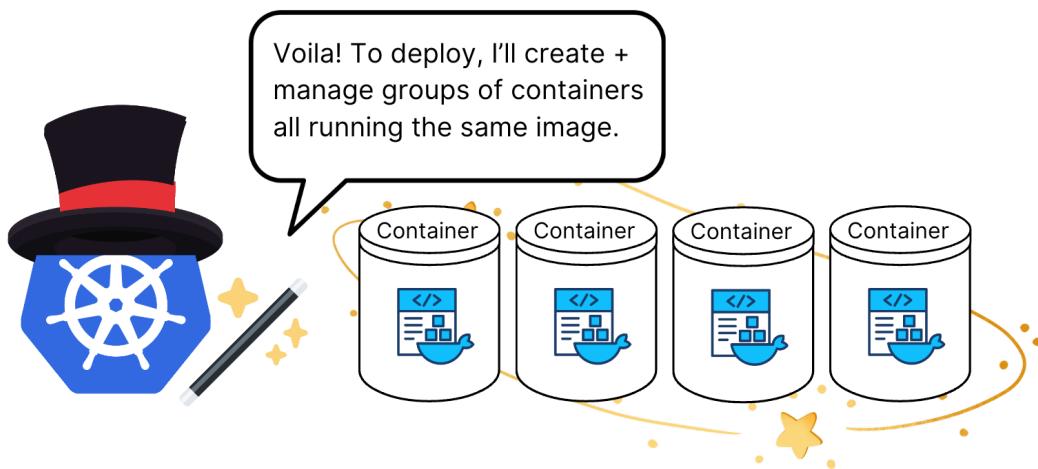
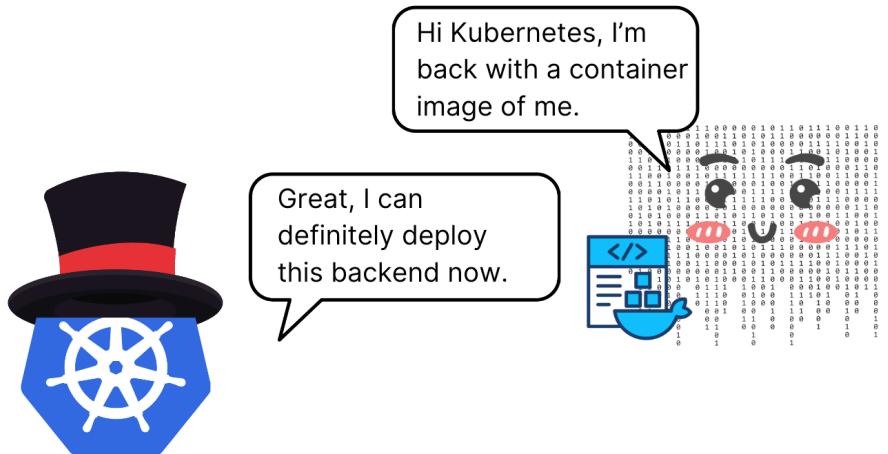
So, when I ran the **docker build** command, I was telling Docker to read those instructions and create a container image based on them. Essentially, I was generating a

portable, self-contained version of the backend that Kubernetes (or any container runtime) could run anywhere.



The container image I built allowed Kubernetes to spin up multiple, identical containers, ensuring my application ran consistently across different environments.

Whether I deployed in development, testing, or production, the app behaved the same way every time because Kubernetes used my image as the source whenever a new instance or node needed to be created.



What do these commands do?

-t Kubernetes-Part-2_dschundru-flask-backend . named my container image Kubernetes-Part-2_dschundru-flask-backend, and the(.) tells Docker to find the Dockerfile in the current directory.

An error....!

```
[ec2-user@ip-172-31-9-43 ~]$ docker build -t Kubernetes-Part-2_dschundru-flask-backend .
-bash: docker: command not found
[ec2-user@ip-172-31-9-43 ~]$ █
```

What is this error about?

Remember what a "command not found" error usually tells you?

Building a container image using Docker won't work as Docker isn't installed in my EC2 instance yet.

Installed and Configured Docker

sudo yum install -y docker

```
[ec2-user@ip-172-31-9-43 ~]$ sudo yum install -y docker
Last metadata expiration check: 11:18:11 ago on Fri Apr 25 05:04:32 2025.
Dependencies resolved.
=====
== Package           Architecture      Version
Repository
=====

```

What is Docker?

Docker is the tool I'm using to **build** a container image of my backend. In general, We'd use Docker to create containers and container images, while Kubernetes coordinates multiple containers (clusters) running the same/related applications.

- Start Docker:

sudo service docker start

```
[ec2-user@ip-172-31-9-43 ~]$ sudo service docker start
Redirecting to /bin/systemctl start docker.service
```

Built the Docker Image

- I ran the command to build my container image again:

```
[ec2-user@ip-172-31-9-43 ~]$ docker build -t Kubernetes-Part-2_dschundru-flask
-backend .
ERROR: permission denied while trying to connect to the Docker daemon socket at unix:///var/run/docker.sock: Get "http://%2Fvar%2Frun%2Fdocker.sock/_ping": dial unix /var/run/docker.sock: connect: permission denied
```

What's the error this time?

The command fails because **ec2-user**, which is the user I've used to access this instance, doesn't have permission to run Docker commands. When Docker was installed, it was set up for the **root** user.

The previous Docker commands I ran worked because they're prefixed with **sudo**, which lets a non-root user run commands with root user rights. But, it's good practice to give my ec2-user permission instead of using **sudo** each time.

Why was I logged in as ec2-user instead of the root user?

I was logged in as **ec2-user** because Amazon Linux AMIs and SSH access use it as the default user.

It's a regular user with permission to run root-level commands using **sudo**. Since Docker needs root access to build images or run containers, I used **sudo** before Docker commands to get the required privileges.

- To confirm which user I was using in the terminal, I ran: **whoami**

```
[ec2-user@ip-172-31-9-43 ~] $ whoami  
ec2-user
```

Added ec2-user to the Docker group:

```
sudo usermod -a -G docker ec2-user
```

What is the Docker group?

The Docker group is a group in Linux systems that gives users the permission to run Docker commands. By default, only the root user can run Docker commands. When we add a user (e.g., ec2-user) to the Docker group, it lets that user run Docker commands without typing sudo every time.

What did this command do?

The command `sudo usermod -a -G docker ec2-user` added the `ec2-user` to the `docker` group.

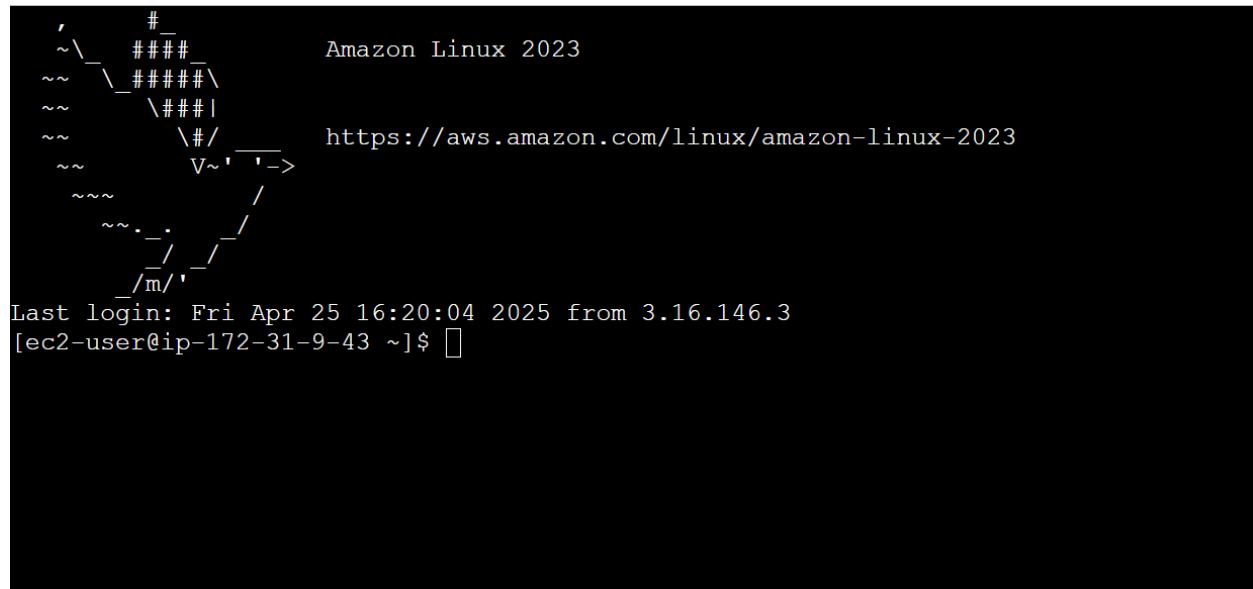
- `usermod` modifies user account settings like their groups, home directory, login name, and more.
- `-a` (append) ensures the user isn't removed from existing groups. Without `-a`, the user would be removed from all groups not listed in the command.
- `-G` specifies which group to add the user to - here, it's `docker`.

This allowed me to run Docker commands without needing `sudo` every time.

- Restarted my **EC2 Instance Connect** session by refreshing my current tab.

Why are we refreshing our tab?

User group changes don't take effect until I start a new session, so reconnecting to my EC2 instance makes sure my `ec2-user` picks up the new docker group permissions.



```

,      #
~\_###_#      Amazon Linux 2023
~~\####\#
~~ \###|
~~   \#/  _--> https://aws.amazon.com/linux/amazon-linux-2023
~~   V~' ,-->
~~~   /
~~ ._. /_/
~~ /m/ ' /_
Last login: Fri Apr 25 16:20:04 2025 from 3.16.146.3
[ec2-user@ip-172-31-9-43 ~]$ 

```

i-04423d205971acb87 (dschundru-eks-instance)

PublicIPs: 3.145.88.208 PrivateIPs: 172.31.9.43

- Make sure my ec2-user has been added to the Docker group:

```
groups ec2-user
```

What does this command do?

This command will list all the groups that ec2-user is a part of.

If I see docker listed, then ec2-user is in the Docker group and can run Docker commands without using sudo.

```
[ec2-user@ip-172-31-9-43 ~]$ groups ec2-user
ec2-user : ec2-user adm wheel systemd-journal docker
[ec2-user@ip-172-31-9-43 ~]$ ]
```

- Yes, I could run the Docker build command again by pressing the **up arrow (↑)** on my keyboard to cycle through my recent terminal commands until the build command appeared:

```
[ec2-user@ip-172-31-9-43 ~]$ docker build -t kubernetes_part2_dschundru-flask-backend .
[+] Building 0.1s (1/1) FINISHED
      docker:default
      0.0s
      0.0s
ERROR: failed to solve: failed to read dockerfile: open Dockerfile: no such file or director
y
[ec2-user@ip-172-31-9-43 ~]$ ]
```

Oof, another error!

Yup! My command won't work because my terminal needs a Dockerfile to build a container image, but it can't seem to find one right now..

What were the results telling me ?

The terminal showed the directories and files in my EC2 instance's root directory.

I saw a folder called `kubernetes_part2_dschundru-flask-backend`, which is the cloned GitHub repository containing the backend code and Dockerfile.

To build the Docker image, I needed to move into the `kubernetes_part2_dschundru-flask-backend` folder, or Docker wouldn't find the Dockerfile needed to create the image.

To **build my Docker image**, I first needed to navigate into the application directory that had the Dockerfile.

I did this by running:

```
cd kubernetes_part2_dschundru-flask-backend
```

- I ran `ls` inside `kubernetes_part2_dschundru-flask-backend` to see the Dockerfile and backend files.

```
[ec2-user@ip-172-31-9-43 kubernetes_part2_dschundru-flask-backend]$ ls
Dockerfile  README.md  app.py  kubernetes_part2_dschundru-flask-backend  requirements.txt
```

- I ran `docker build -t kubernetes_part2_dschundru-flask-backend .` to build the Docker image again.

```
[ec2-user@ip-172-31-9-43 kubernetes_part2_dschundru-flask-backend]$ docker build -t kubernetes_part2_dschundru-flask-backend .
[*] Building 11.1s (10/10) FINISHED
      docker:default
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 281B
=> [internal] load metadata for docker.io/library/python:3.9-alpine
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [1/5] FROM docker.io/library/python:3.9-alpine@sha256:c549d512f8a56f7dbf15032c0b21799f022118d4b7254 1.7s
=> => resolve docker.io/library/python:3.9-alpine@sha256:c549d512f8a56f7dbf15032c0b21799f022118d4b7254 0.0s
=> => sha256:f1e232174bc91741fd3da96d85011092101a032a93a388b79e96c2d5c870 3.64MB / 3.64MB 0.2s
=> => sha256:31050cb47a0204aa139821ee500ed6b13dc7142d89b12154f9a2d2efba8a6ab7 460.18kB / 460.18kB 0.3s
=> => sha256:374b62c84664db7f5059aa54735d1e7921d3350b69515a1c9651201795807c3d 14.87MB / 14.87MB 0.4s
=> => sha256:c549d512f8a56f7dbf15032c0b21799f022118d4b72542b8d85e2eae350fcfd7 10.29kB / 10.29kB 0.0s
=> => sha256:920c6dde0859e15c81b84b28964b7a47771d593fb39c6f8b050f 1.73kB / 1.73kB 0.0s
=> => sha256:3f22234415a3570fc933cd711b6ba7b5905d0c6367cf747af84dbf9fbee42b10 5.08kB / 5.08kB 0.0s
=> => extracting sha256:f1e232174bc91741fd3da96d85011092101a032a93a388b79e96c2d5c870 0.3s
=> => sha256:4facdbdcc8a37a46035340540047c8eed35e9b8dfd033632e0438d03f82d021a 250B / 250B 0.3s
=> => extracting sha256:31050cb47a0204aa139821ee500ed6b13dc7142d89b12154f9a2d2efba8a6ab7 0.1s
=> => extracting sha256:374b62c84664db7f5059aa54735d1e7921d3350b69515a1c9651201795807c3d 0.8s
=> => extracting sha256:4facdbdcc8a37a46035340540047c8eed35e9b8dfd033632e0438d03f82d021a 0.0s
=> [internal] load build context 0.0s
=> => transferring context: 83.03kB 0.1s
=> => transferring context: 83.03kB 0.0s
```

i-04423d205971acb87 (dschundru-eks-instance)

Public IPs: 3.145.88.208 Private IPs: 172.31.9.43



Step 4: Pushed My Container Image to Amazon ECR

Now that my container image is all built, where should Kubernetes find my container image?

Container registries, like **Amazon Elastic Container Registry (ECR)**, are storage spaces for container images. They give container images somewhere to live so they can be accessed by Kubernetes/other services.

Why am I using Docker and ECR?

I'm using Docker to package up the backend and then pushing that container image to Amazon ECR (Elastic Container Registry). When it's time for Kubernetes to deploy my application, Kubernetes can just pull the image directly from ECR.

This setup is the standard way for deploying containerized apps with Kubernetes (instead of deploying code on my local machine) because it helps with consistency and scaling.

In this step, I:

- Stored the Docker image of my backend in ECR.

Created a new Amazon ECR repository using this command:

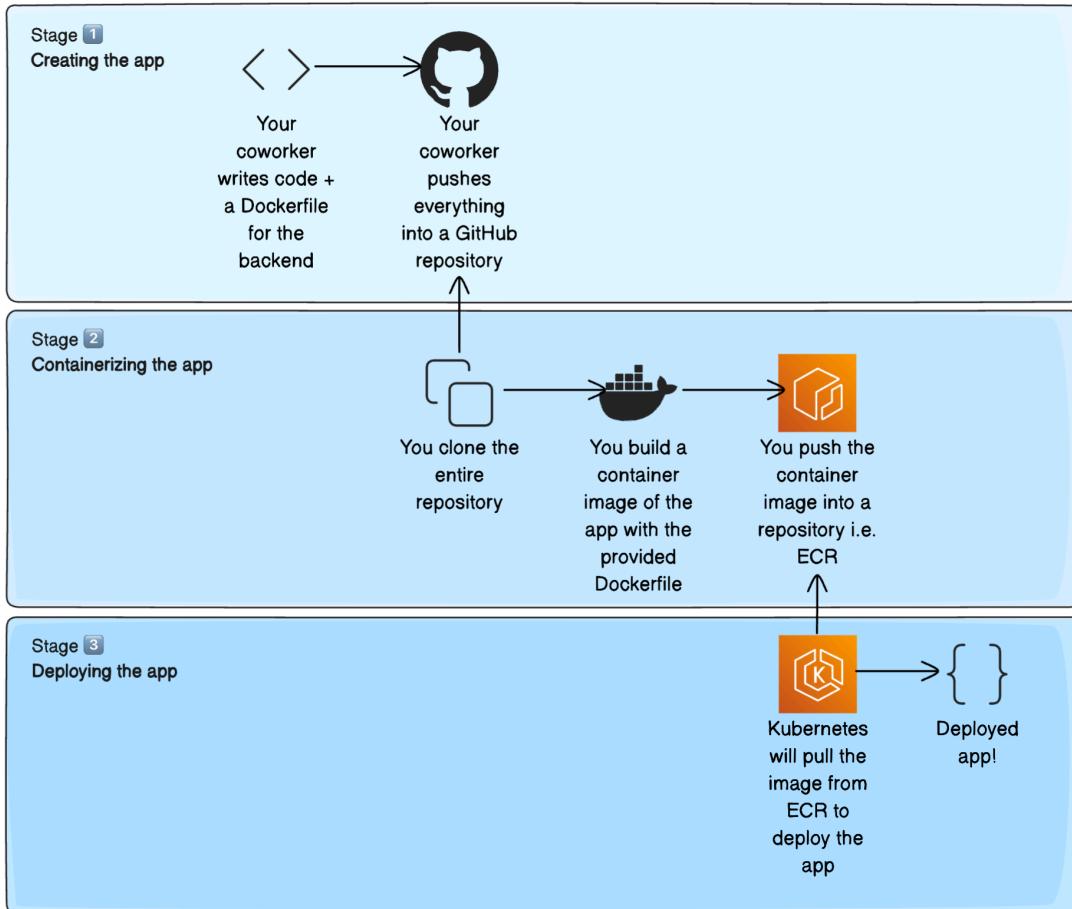
```
aws ecr create-repository \
--repository-name kubernetes_part2_dschundru-flask-backend \
--image-scanning-configuration scanOnPush=true \
```

Why am I using ECR?

Amazon ECR (Elastic Container Registry) is a container registry service by AWS, which means I use it to securely store, share, and deploy container images.

ECR is an excellent choice for storing my container image because it's an AWS service, which lets Elastic Kubernetes Service (EKS) deploy my container image with minimal authentication setup.

The process from writing code to deploying an app



```
[ec2-user@ip-172-31-9-43 ~]$ aws ecr create-repository \
--repository-name kubernetes_part2_dschundru-flask-backend \
--image-scanning-configuration scanOnPush=true \
{
  "repository": {
    "repositoryArn": "arn:aws:ecr:us-east-2:329599620395:repository/kubernetes_part2_dschundru-flask-backend",
    "registryId": "329599620395",
    "repositoryName": "kubernetes_part2_dschundru-flask-backend",
    "repositoryUri": "329599620395.dkr.ecr.us-east-2.amazonaws.com/kubernetes_part2_dschundru-flask-backend",
    "createdAt": "2025-04-25T20:50:08.857000+00:00",
    "imageTagMutability": "MUTABLE",
    "imageScanningConfiguration": {
      "scanOnPush": true
    },
    "encryptionConfiguration": {
      "encryptionType": "AES256"
    }
  }
}[ec2-user@ip-172-31-9-43 ~]$ █
```

i-04423d205971acb87 (dschundru-eks-instance)

PublicIPs: 3.145.88.208 PrivateIPs: 172.31.9.43

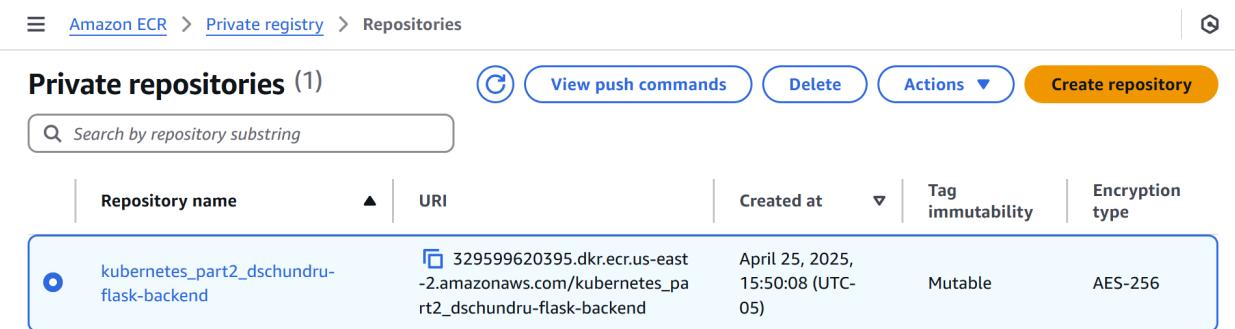
What does the terminal's response mean?

This response confirms that my ECR repository has been created - it's ready for me to push Docker images!

Here's a breakdown of the terminal response

- **repositoryArn:** The Amazon Resource Name (ARN) i.e. unique ID for my ECR repository.
- **repositoryUri:** This is the URL I'll use to push and pull container images. It shows where my images will be stored.
- **repositoryName:** The name I've given to my repository in this case, kubernetes_part2_dschundru-flask-backend.
- **imageTagMutability:** Whether image tags are mutable or immutable. "MUTABLE" means I can overwrite which image has a tag e.g. the latest tag can be taken by a newer image at any time.
- **imageScanningConfiguration:** Whether images will be scanned for vulnerabilities when pushed.
- **encryptionConfiguration:** My images are encrypted using AES256 for security.

I opened a new tab, go to the ECR console, and confirmed that a new repository called **kubernetes_part2_dschundru-flask-backend** was created.



The screenshot shows the Amazon ECR console interface. At the top, there are navigation links: 'Amazon ECR' > 'Private registry' > 'Repositories'. Below this, a header bar includes 'Private repositories (1)', a search bar ('Search by repository substring'), and buttons for 'View push commands', 'Delete', 'Actions', and 'Create repository'. The main table displays one repository row:

Repository name	URI	Created at	Tag immutability	Encryption type
kubernetes_part2_dschundru-flask-backend	329599620395.dkr.ecr.us-east-2.amazonaws.com/kubernetes_part2_dschundru-flask-backend	April 25, 2025, 15:50:08 (UTC-05)	Mutable	AES-256

Nice, my Amazon ECR repository was live. I was ready to push my container image into ECR!

Pushed my container image to ECR

- I selected my new repository and clicked **View push commands**.

What is a push command?

A push command in Amazon ECR is used to upload our Docker container images to an ECR repository.

The screenshot shows the AWS ECR console interface. At the top, there's a breadcrumb navigation: Amazon ECR > Private registry > Repositories > kubernetes_part2_dschundru-flask-backend. Below this, a header bar includes 'Images (0)', a search bar ('Search artifacts'), and buttons for 'Delete', 'Details', 'Scan', and 'View push commands'. The 'View push commands' button is highlighted with a yellow background. A pagination indicator shows '1' of 1 page. On the far right, there's a gear icon for settings.

- I copied the first command.

The screenshot shows a modal window titled 'Push commands for kubernetes_part2_dschundru-flask-backend'. It has two tabs: 'macOS / Linux' (which is selected) and 'Windows'. A note at the top of the modal reads: 'Make sure that you have the latest version of the AWS CLI and Docker installed. For more information, see [Getting Started with Amazon ECR](#)'. Below this, instructions say to use the following steps to authenticate and push an image to your repository. It also includes a note about Registry Authentication and a warning about unencrypted password storage.

1. Retrieve an authentication token and authenticate your Docker client to your registry. Use the AWS CLI:

```
aws ecr get-login-password --region us-east-2 | docker login --username AWS --password-stdin 329599620395.dkr.ecr.us-east-2.amazonaws.com
```

Note: If you receive an error using the AWS CLI, make sure that you have the latest version of the AWS CLI and Docker installed.

- I ran the command in my EC2 Instance Connect window.

```
[ec2-user@ip-172-31-9-43 ~]$ aws ecr get-login-password --region us-east-2 | docker login --username AWS --password-stdin 329599620395.dkr.ecr.us-east-2.amazonaws.com
WARNING! Your password will be stored unencrypted in /home/ec2-user/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store
Login Succeeded
```

I headed back to my ECR console's push commands window. Since I had already built my container image, I skipped the second command.

I copied the **last two** push commands.

2. Build your Docker image using the following command. For information on building a Docker file from scratch see the instructions [here](#). You can skip this step if your image is already built:

```
 docker build -t kubernetes_part2_dschundru-flask-backend .
```

3. After the build completes, tag your image so you can push the image to this repository:

```
 docker tag kubernetes_part2_dschundru-flask-backend:latest 329599620395.dkr.ecr.us-east-2.amazonaws.com/kubernetes_part2_dschundru-flask-backend:latest
```

4. Run the following command to push this image to your newly created AWS repository:

```
 docker push 329599620395.dkr.ecr.us-east-2.amazonaws.com/kubernetes_part2_dschundru-flask-backend:latest
```

- I ran the last two in my EC2 Instance Connect terminal to tag and push my container image.

```
[ec2-user@ip-172-31-9-43 ~]$ docker tag kubernetes_part2_dschundru-flask-backend:latest 329599620395.dkr.ecr.us-east-2.amazonaws.com/kubernetes_part2_dschundru-flask-backend:latest
[ec2-user@ip-172-31-9-43 ~]$ docker push 329599620395.dkr.ecr.us-east-2.amazonaws.com/kubernetes_part2_dschundru-flask-backend:latest
The push refers to repository [329599620395.dkr.ecr.us-east-2.amazonaws.com/kubernetes_part2_dschundru-flask-backend]
78744ade330f: Pushed
9ad59d572c60: Pushed
340dfe38e033: Pushed
acd9d20cc5d9: Pushed
145d798cd760: Pushed
55e198163324: Pushed
98478a764fb3: Pushed
08000c18d16d: Pushed
latest: digest: sha256:5971ee65c4dfe01556d2a683800c01f09dd9609c273c368862b9211dfee2001d size: 1991
[ec2-user@ip-172-31-9-43 ~]$
```

What do tagging and pushing the image do?

Since my ECR repository can hold many versions of the same container image, tags help me keep things organized. Tagging my Docker image is like giving it a nickname so I can easily refer to a specific version.

Here, I'm **tagging** my Docker image with the latest so Kubernetes knows where it can find the right container image version when it's time to deploy. **Pushing** uploads the tagged image to a remote repository. In my case, I've just uploaded my container image to my ECR repository!

- I headed back to the ECR console, closed the push commands window, and clicked the refresh button to update my console.

The screenshot shows the Amazon ECR console interface. At the top, there's a breadcrumb navigation: Amazon ECR > Private registry > Repositories > kubernetes_part2_dschundru-flask-backend. Below the navigation, it says "Images (0)". There are four buttons: "Delete", "Details", "Scan", and a yellow "View push commands" button. A search bar with placeholder text "Search artifacts" is below the buttons. On the right, there are navigation arrows and a gear icon.

- Confirmed that a new container image is in my console now.

The screenshot shows the Amazon ECR console interface after pushing an image. It displays "Images (1)". The table has columns: Image tag, Artifact type, Pushed at, Size (MB), Image URI, Digest, and Last recorded pull time. One row is shown:

Image tag	Artifact type	Pushed at	Size (MB)	Image URI	Digest	Last recorded pull time
latest	Image	April 25, 2025, 16:19:04 (UTC-05)	39.17	<input type="button" value="Copy URI"/>	<input type="button" value="sha256:5971ee6..."/>	April 25, 2025, 16:20:02 (UTC-05)

Does Kubernetes only run containerized applications from a repository?

No, Kubernetes doesn't have to pull images from a repository but using one like ECR makes things easier.

- With a registry**, Kubernetes can pull the latest image automatically across all nodes.
- Without a registry**, I would have to manually load and update the image on every node.

Secret Mission: Dive into the Backend Code

To explore the backend repository I cloned. By the end of this project, I'll truly know what was inside the Docker image I built and pushed in this project.

In this secret mission, I:

- Opened the backend repository on GitHub.
- Explore the three files that make up the backend.
- Showcased my secret mission in my project documentation.

My team member sent me a link to Search Hacker News and asked me to search for something.

- I typed a keyword and hit enter.

The backend received my request, found the matching articles, and sent them back to me so I could see the results.

What is Search Hacker News?

Hacker News is an online platform and community where we can read and discuss tech-related news and industry trends.

Search Hacker News is the Hacker News platform's search engine. I'd use it to quickly find relevant content based on my search terms.

- I opened this link to Search Hacker News in a new tab, which already has the search query aws pre-filled: <https://hn.algolia.com/?q=aws>

The screenshot shows the Hacker News search interface with the query 'aws' entered. The search results page displays 21,609 results found in 0.006 seconds. The results are a list of links, titles, and brief descriptions related to AWS, such as 'AWS services explained in one line each', 'AWS us-east-1 outage', and 'AWS forked my project and launched it as its own service'. The interface includes a navigation bar with 'Search Stories by Popularity for All time', a search bar, and algolia search branding.

My coworker explained that the backend app was a tool that:

- Took my input (for example, "aws")
- Ran it as a search query in Hacker News Search
- Processed the search results and formatted them into JSON data

Even though I hadn't seen the backend's JSON output yet (because I hadn't deployed it), using Search Hacker News gave me a good idea of the kind of data the backend would produce.

Next part (answering "How does the backend code do all this?" simply:

👉 The backend code sent my input to Hacker News Search, collected the results, cleaned them up, and organized them into a neat JSON format.

ExploreD the Code

- Open the [backend GitHub repository](#) in my browser.

kubernetes_part2_dschundru-flask-backend Public

main 1 Branch 0 Tags

Go to file Add file Code

DeviSuhithaChundru Add files via upload 4a7f61d · 4 hours ago 3 Commits

Dockerfile	Add files via upload	4 hours ago
README.md	Add files via upload	4 hours ago
app.py	Add files via upload	4 hours ago
requirements.txt	Add files via upload	4 hours ago

What's in this GitHub repository?

This GitHub repository holds all the source code for your app's backend.

There are three main characters in your backend app's code:

- **requirements.txt** lists the dependencies my app needs.
- The **Dockerfile** provides a set of instructions for building a Docker image for my app.
- **app.py** is the core of my backend. It contains the actual code that determines what my app does and the kind of responses it sends back to the client.

Look at **requirements.txt**. This file lists the dependencies my application needs to run properly.

[kubernetes_part2_dschundru-flask-backend / requirements.txt](#)

DeviSuhithaChundru Add files via upload

Code Blame 4 lines (4 loc) · 65 Bytes

Code 55% faster with GitHub Copilot

```

1  Flask==2.1.3
2  flask-restx==0.5.1
3  requests==2.28.1
4  werkzeug==2.1.2

```

Can I break down the dependencies in requirements.txt?

- Flask==2.1.3: **Flask** is the web framework used to build the backend code, which I would see in **app.py**.
- flask-restx==0.5.1: My app creates an **API** using an extension **Flask-RESTX** so that users or other applications can make requests to our backend.
- requests==2.28.1: The **Requests** library is used to get data from the Hacker News **API**.
- werkzeug==2.1.2: **Werkzeug** helps Flask handle application-level routing. For example, when a user/service makes a request to my backend, Flask uses its routing system (powered by Werkzeug) to direct that request to the right.

Selected the **Dockerfile**.

[kubernetes_part2_dschundru-flask-backend / Dockerfile](#) 



DeviSuhithaChuntru Add files via upload

[Code](#)

[Blame](#)

7 lines (7 loc) • 183 Bytes



Code 55% faster with GitHub Copilot

```

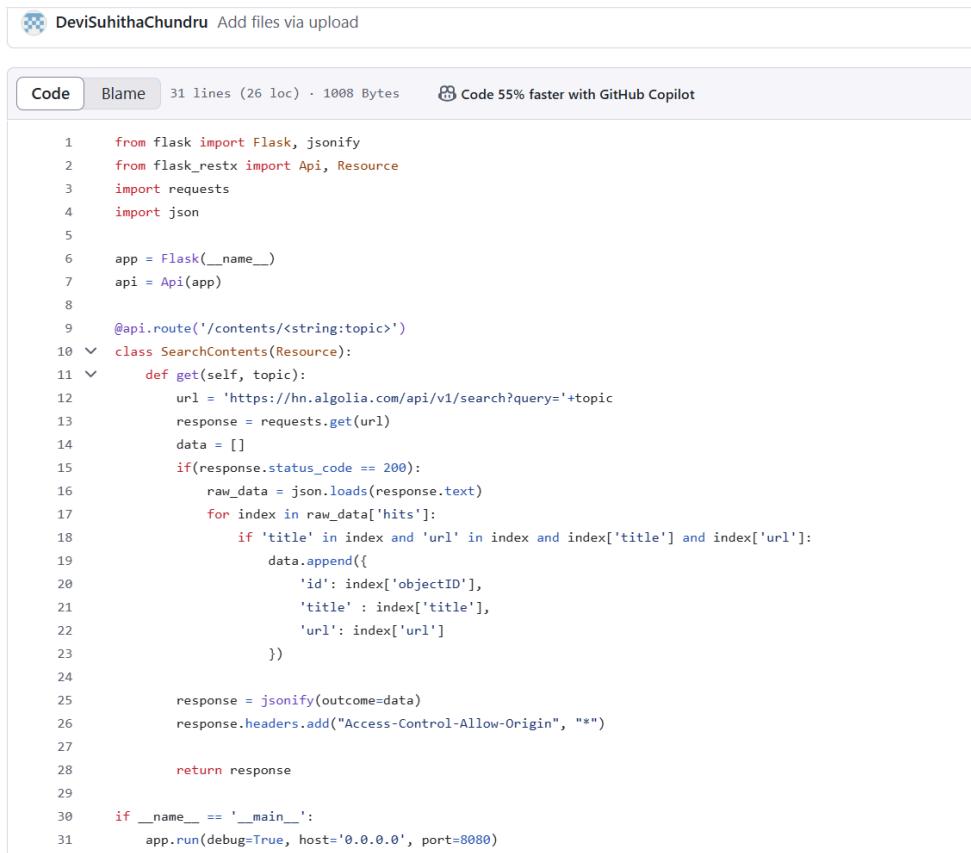
1  FROM python:3.9-alpine
2  LABEL Author="Devi Suhitha Chuntru"
3  WORKDIR /app
4  COPY requirements.txt requirements.txt
5  RUN pip3 install -r requirements.txt
6  COPY . .
7  CMD ["python3", "app.py"]

```

Can I break down the commands in this Dockerfile?

- **FROM python:3.9-alpine** — I set up the environment by using Python 3.9 on Alpine Linux, which kept my app small and fast.
- **LABEL Author="NextWork"** — I added author information to the image.
- **WORKDIR /app** — I set **/app** as the working directory where the next commands would run.
- **COPY requirements.txt requirements.txt** — I copied the **requirements.txt** file from my local machine into the container.
- **RUN pip3 install -r requirements.txt** — I installed all the dependencies listed in **requirements.txt**.
- **COPY . .** — I copied all my project files (like **requirements.txt** and **app.py**) into the container's **/app** directory.
- **CMD ["python3", "app.py"]** — I told the container to run **python3 app.py** to start my Flask app.

Now selected **app.py**.



The screenshot shows a GitHub code editor interface. At the top, there is a user icon and the text "DeviSuhithaChundru Add files via upload". Below this is a navigation bar with tabs: "Code" (which is selected), "Blame", and "31 lines (26 loc) · 1008 Bytes". There is also a link "Code 55% faster with GitHub Copilot". The main area contains the following Python code:

```

1  from flask import Flask, jsonify
2  from flask_restx import Api, Resource
3  import requests
4  import json
5
6  app = Flask(__name__)
7  api = Api(app)
8
9  @api.route('/contents/<string:topic>')
10 class SearchContents(Resource):
11     def get(self, topic):
12         url = 'https://hn.algolia.com/api/v1/search?query=' + topic
13         response = requests.get(url)
14         data = []
15         if response.status_code == 200:
16             raw_data = json.loads(response.text)
17             for index in raw_data['hits']:
18                 if 'title' in index and 'url' in index and index['title'] and index['url']:
19                     data.append({
20                         'id': index['objectID'],
21                         'title': index['title'],
22                         'url': index['url']
23                     })
24
25         response = jsonify(outcome=data)
26         response.headers.add("Access-Control-Allow-Origin", "*")
27
28     return response
29
30 if __name__ == '__main__':
31     app.run(debug=True, host='0.0.0.0', port=8080)

```

What's happening in app.py?

app.py is the main code for my backend. It has three main parts:

- **Setting up the app and routing:** The code starts by importing the Flask framework and tools for creating an API. It defines a route (/contents/<topic>) that directs incoming requests to the SearchContents class. This means users visiting my backend will need to visit a URL that ends with /contents/<topic> to see any results
- **Fetching data:** In the **SearchContents** class, the get() method extracts the topic from the URL, sends a request to the Hacker News Search API to find related content, and collects important details like id, title, and url.
- **Sending the response:** The collected data is turned into a formatted JSON response.

The app.py file contains three main parts: Installing dependencies, formatting the data into JSON data, passing the formatted data back to the user/requester.

What is an API?

An API (Application Programming Interface) is like a bridge that lets different programs talk to each other. It lets one app to ask for data or services from another and get a response.

For example, when my backend app runs a search using the Hacker News Search API, it sends a request, gets data back, and processes it into JSON format to return to the user. This helps different parts of software work together easily.

In my own backend app, **I also create MY own API** using Flask. This API takes user input, connects to the Hacker News API to get the data, processes that data, and then sends it back as JSON.

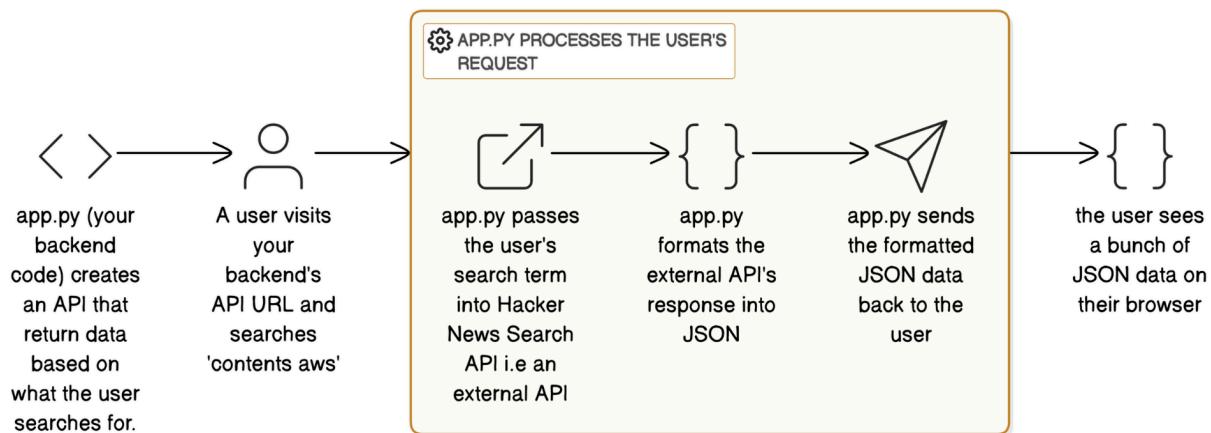
APIs make it easy for my backend to collect data from Hacker News and then share information back to my users and other services.

That's the backend code investigation all done!

Let's wrap up with a quick summary:

1. My app's backend **fetches data** based on a search topic I enter.
2. The backend code uses **Flask** to connect with an **external API** (Hacker News Search API) and process the data.
3. The backend then sends the data back as formatted JSON.

How app.py processes your requests



After reviewing the app's backend code, I've learnt that the app functions by extracting data from another API, but I also have my own API in app.py that's responsible for others wanting access to my service!

Thank you !!